**Data Structure**

1. Recursive method: must be structured to handle both the base case and the recursive case
2. Running Time Analysis

Asymptotic analysis: Look at growth of T(n) (running time of input n) as n→∞. Ignore low-order terms and leading constants

1. Big-O notation

if constants c and n0 such that 0 ≤ O(g(n)) ≤ c\*g(n) n0 ≤ n

Always use the tightest one -> 2n2 = O(n3) or 2n2 = O(n4) but not 2n2 = O(n)

1. merge sort: O(n log n) Insertion sort: (n2)
2. Inner Class: Class within another class

Outer class and Inner class can access the instance variables of each other directly.

**Linked list** (Dynamic data type)

public class MyList {

private MyNode list; //instance variable for the reference to the first node

// (Inner class) hidden from outside MyList

private class MyNode {

Object obj; //instance variable for the object it carries

MyNode next; // instance variable for the reference to the next node

MyNode(Object item) {

obj = item;

next = null;

}

}

//add a node at the front

public void addtoFront(MyNode newNode){

newNode.next = list;

list = newNode;}

//Find an object

public boolean contains(Object item) {

MyNode current = list;

while (current != null) {

if (current.obj.equals(item))

return true;

current = current.next;

}

return false;}

//Remove the front node

public void removeHead() {

if (list == null)

return;

list = list.next;

}

**Doubly linked list**

//remove one node in the doubly linked list

//the previous node of the first node and the next node of the last node is null

public void remove(MyNode node){

if (node.prev == null){

list = node.next;

node.next.prev = null;

}

else if (node.next == null)

node.prev.next = null;

else{

node.prev.next = node.next;

node.next.prev = node.prev;

}

**Queues**

(FIFO): adds items only to the rear of the list and removes them only from the front

The items can be kept in a linked list(more efficient) or an arraylist.

Operations:

enqueue - add an item to the rear of the queue

dequeue - remove an item from the front of the queue

empty - returns true if the queue is empty

size – returns the size of the queue

public class MyQueue {

private ArrayList<Object> queue;

public MyQueue () {

queue = new ArrayList<Object>();

}

public boolean empty() {

return queue.isEmpty();

}

public void enqueue(Object item) {

queue.add(item);

}

public Object dequeue () {

if (queue.isEmpty()) return null;

return queue.remove(0);

}

}

**Stacks**

(LIFO): Items are added and removed from only one end of a stack

Operations:

push - add an item to the top of the stack

pop - remove an item from the top of the stack

peek/top - retrieves the top item without removing it

empty - returns true if the stack is empty

public class MyStack {

private ArrayList<Object> stack;

public MyStack () {

stack = new ArrayList<Object>();

}

public boolean empty(){

return (stack.isEmpty());

}

public void push(Object item){

stack.add(0,item);

}

public Object pop(){

if (stack.isEmpty()) return null;

return stack.remove(0);

}

}

**Trees**:

Non-linear data structure

Exactly one path connecting two nodes together (N nodes -> N-1 links)

1. Binary trees:
2. **Siblings**: Nodes having the same parent
3. **Leaf**: A node that has no child <-> **internal node**: A node that has one or two children
4. **Subtree**: The node and all nodes descending from it
5. **Ancestor**: nodes lie on the path leading to the root <-> descendant: nodes lie in the subtree
6. **Height of Tree**: the number of edges on the longest path from the root to a leaf

//establish a binary tree

public class BinTree {

private BinTreeNode root;

public BinTree() {

root = null;

}

}

public class BinTreeNode {

private BinTreeNode parent, left, right;

private int item;

public BinTreeNode(int item) {

this.item = item;

parent = left = right = null;

}

public boolean isRoot(){

return (parent == null);

}

public boolean isLeftChild(){

if (parent == null) //always check NullPointerException

return false;

return (this == parent.left);

}

// return the distance from root to a node

public int distFromRoot(){

if (parent == null){

return 0;

else

return parent.distFromRoot()+1;

}

}

//remove a node

public void removeMyself(){

//is a leaf

if (left == null && right == null){

if (parent == null)

root = null;

else

parent.left = parent.right == null;

//only have left child

else if (left != null && right == null){

if (parent == null){

left.parent = null;

root = left;

}

else{

left.parent = parent;

if (isLeftChild())

parent.left = left;

else

parent.right = left;

}

else{

MyNode node = right;

//find the leftmost node of right subtree

while(node.left != null)

node = node.left;

if (parent == null){

root = node;

node.parent = null;

node.left = left;

node.right = right;

}

else{

if (node.right != null){

node.right.parent = node.parent;

if node.isLeftChild()

node.parent.left = node.right;

else

node.parent.right = node.right;

}

node.parent = parent;

node.left = left;

node.right = right;

if (isLeftChild())

parent.left = node;

else

parent.right = node;

}

}

}

//return the size of binary tree -> size(node) = size(left\_subtree)+size(right\_subtree)+1

public int size(){

int leftSize = 0;

int rightSize = 0;

if (left != null)

leftSize = left.size();

if (right != null)

rightSize = right.size();

return leftSize+rightSize+1;

}

1. Tree traversal :
2. Pre-order: root-left-right;
3. In-order: left-root-right;
4. Post-order: left-right-root

Public void inOrderTraversal(){

if (left != null)

left.inOrderTraversal();

System.out.print(item + ” “);

if (right != null)

right.inOrderTraversal();

}

1. Ordered-tree: in-order traversal presents the ascending order of the objects
2. Insertion(can use recursion):
3. if the root is null, make the new element to be the root
4. If the item being added is less than or equal to the root, add on the left subtree (If there is no left subtree, item is added as the left child)
5. If the item being added is larger than the root, add on the right subtree (If there is no right subtree, item is added as the right child)

Deletion:

1. If the node is a leaf, simply remove it
2. If the node has a single child, promote the child
3. If the node has two children, find the leftmost child on the right subtree to replace it
4. **Heap**: complete binary tree: each level is filled except possibly the bottommost level, which must be packed to the left if not completely filled.

The value stored at a node ≤ the value(s) stored at the child(ren)

The root of a heap contains the minimum number of the whole heap

1. Accessing the minimum number: constant time
2. Removing the minimum number: O(log2n) -> can only remove the smallest one
3. copy the last node to the root
4. Re-heapify the resultant tree: keep swapping the root with the smaller child until the parent is smaller than both children
5. Adding a new number:
6. Add the new number in the bottommost level
7. Re-heapify the resultant tree: keep swapping if parent is larger (poping up)
8. Heap sort: O(nlog2n)
9. Insert each number one by one to the heap: O(nlog2n)
10. Remove the minimum number from the heap one by one until the heap is empty: O(nlog2n)
11. Array implementation: (elements are stored level-by-level)

For element arr[i], left child -> arr[2\*i+1], right child -> arr[2\*i+2], parent -> arr[(i-1)/2].

1. **Huffman Encoding**: Ensures that the minimum number of bits is used to encode a passage using a binary code (string of 0’s and 1’s)

-> Based on frequency: the number of occurrence of each character is known

-> The larger the frequency, the shorter the code

(1) Process:

a. In the beginning, each node that represents a certain character forms its own subtree. The subtrees are sorted according to their frequencies carried in the root in ascending order.

b. Pick the two subtrees with the smallest frequencies and put them as the left child and right child of a new subtree. Keeps on going until all the subtrees are connected to form a single tree.

c. Either all edges leading to a left child are assigned to 1 or 0.

d. The code of the character is the bit pattern obtained by traversing the edges from the root to the leaf node representing the character.

(2) decode: traverse the tree to identify the characters

1. **AVL tree**: heights of the left and right subtrees differ by at most 1

The height of an empty tree is -1

1. Insert a new node: put it in the right position according to the order -> if the AVL tree property is violated, rebalance the tree through rotations:

(heights become imbalanced at node α)

1. left subtree of the left child too tall -> Single rotation at α (promote left child as parent)
2. right subtree of the right child too tall -> Single rotation at α (promote right child as parent)
3. left subtree of the right child is too tall -> double rotation (right child then α)
4. right subtree of the left child is too tall -> double rotation (left child then α)

类中的方法只针对类的实例变量